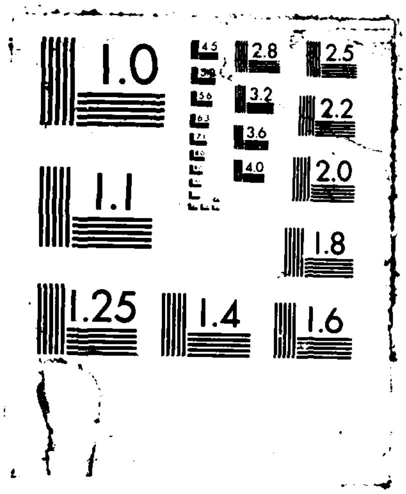
AD-A189 376 AGGREGATES IN THE TEMPORAL QUERY LANGUAGE TQUEL(U) 1/1
NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER
SCIENCE R SNODGRASS ET AL. 27 JUL 87 UNC-TEMPIS-16
UNCLASSIFIED N00014-86-K-0680 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

4

The TEMPIS Project

Aggregates in the Temporal Query Language TQuel

Richard Snodgrass, Santiago Gomez, and Ed McKenize

July 27, 1987

Contract N00014-86-K-0680

Abstract

This paper defines aggregates in the temporal query language TQuel and provides their formal semantics in the tuple relational calculus. A formal semantics for Quel aggregates is defined in the process. Multiple aggregates; aggregates appearing in the where, when, valid, and as-of clauses; nested aggregation; and instantaneous, cumulative, moving window, and unique variants are supported. These aggregates provide a rich set of statistical functions that range over time, while requiring minimal additions to TQuel and its semantics.

TEMPIS Document No. 16

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

Table of Contents

Table of Examples

Aggregate operators in query languages compute a scalar value from a collection of tuples in a relational database. Most commercially available relational database management systems (DBMSs) provide several aggregate operations [Date 1983, IBM 1981, Ullman 1982]. Recently attention has been focussed on *temporal databases* (TDBs) that represent the progression of states of an enterprise over time. We have developed a new language, TQuel (*T*emporal *QUE*ry *L*anguage), to query a TDB [Snodgrass 1987]. TQuel is a derivative of Quel [Held et al. 1975], the query language for the Ingres DBMS [Stonebraker et al. 1976]. TQuel was designed to be a minimal extension, both syntactically and semantically, for that language. Since Quel is fairly comprehensive in its support of aggregates, a goal in the TQuel design was to extend those aggregates to operate over temporal relations.

This paper defines and formalizes aggregates in TQuel. We begin by developing a formal semantics for Quel aggregates. An intuitive introduction to the TQuel aggregates is given in Section 2. Section 3 is devoted to a formal semantics of TQuel aggregates. The final section compares TQuel aggregates with those of several other query languages supporting time. Throughout the paper, a fixed-width font is used for operators in the query language (e.g., `count`); a bold, fixed-width font is used for keywords (e.g., **`year`**); and italics is used for functions in the formal semantics (e.g., *count*).

1. Aggregates In Quel

In this section we present a complete semantics for the Quel aggregates, as a convenient point of reference for the TQuel semantics to be developed in Section 3. An informal specification for aggregates is given, followed by a formal semantics of the retrieve statement with aggregates in the Quel language.

1.1. Informal Specification of Quel Aggregates

The Quel operations for aggregation are

`count`	The number of values that exist for a given attribute in a relation. Since every attribute has exactly one value in each tuple, this operator yields the same result on all attributes of a relation.
`any`	An indicator of whether there exists at least one tuple in a relation. It returns a 1 if the relation is non-empty and 0 otherwise.
`sum`	The sum of the values present for a given attribute. This operator can be computed only on a numeric attribute.
`avg`	The average, or arithmetic mean, of the values present for a given attribute. The average is defined in the usual way, i.e. the sum divided by the count. Because of this dependency upon sum, `avg` is also an operator on numeric attributes only.
`min`	The smallest of the values present for a given attribute. For an alphanumeric attribute, the alphabetical ordering is used to determine the smallest element.

max The largest of the values present for a given attribute. For an alphanumeric attribute, the alphabetical ordering is used to determine the largest element.

These operators can be used in two types of aggregation:

(a) *Scalar aggregates*, yielding a *single value* as the result.

(b) *Aggregate functions*, producing several values determined by calculating the aggregate over a subset of the relation. Each subset consists of the tuples such that the contents of one or more attributes grouped in a by-list are the same. Hence the result of an aggregate function is a *relation* whose number of tuples equals the number of different values in the by-list.

While scalar aggregates are independent of the query in which they are nested, aggregate functions are not. Since each value computed by such a function carries information on part of a relation, tuple variables in the by-list must be linked to the corresponding tuple variables, if any, in the *outer query* – that is, they should refer to the same part of the relation. (The *inner query*, as opposed to the outer query, is the one consisting of the attribute to be aggregated, the by-list, and the inner where clause.)

By their very nature, both scalar aggregates and aggregate functions operate on the entire relation. However, they can be *locally* restricted via a where clause to operate only on certain tuples of the relation. The local or inner where clause is processed separately from the outer one of the query. We first show an aggregate function, followed by a scalar aggregate.

EXAMPLE. Suppose the relation *Faculty* holds relevant data, say name, rank and salary, about the professors in a university department:

Faculty(Name, Rank, Salary):

Name	*Rank*	*Salary*
Tom	Assistant	23000
Merrie	Assistant	25000
Jane	Associate	33000

```
range of f is Faculty
retrieve (f.Rank, NumInRank = count(f.Name by f.Rank))
```

Example 1: How many faculty members are there in each rank?

The range statement declares a tuple variable `f` that will be associated with the *Faculty* relation. The retrieve statement contains the target list of attributes to be derived for the output relation, in this case, *Rank* and *NumInRank:*

Rank	*NumInRank*
Assistant	2
Associate	1

The output relation contains as many tuples as actual values exist in the by-list. If there had been no by-list, *NumInRank* would be 3 in all the derived tuples. ∎

Aggregation performed over the set of strictly different values in an attribute is called unique aggregation. Quel supports three unique aggregates: `countU`, `sumU`, and `avgU`. Unique versions of `any`, `max` and `min` are not necessary.

EXAMPLE. This example illustrates multiple scalar aggregates and unique aggregation.

```
range of f is Faculty
retrieve (NumFaculty = count(f.Name), NumRanks = countU(f.Rank))
```

Example 2: How many faculty members and different ranks are there?

The result is a single tuple:

NumFaculty	*NumRanks*
3	2

∎

1.2. Semantics of the Quel Retrieve Statement

A tuple relational calculus semantics for Quel statements without aggregates was defined by Ullman [Ullman 1982] and is reviewed here. Although values in a target list can be expressions, rather than simply attributes, we ignore that detail in this paper for simplicity of notation. Thus the skeletal Quel statement is

```
range of $t_1$ is $R_1$
...
range of $t_k$ is $R_k$
retrieve ($t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r}$)
          where $\psi$
```

in which

$$1 \le i_1 \le k, \ldots, 1 \le i_r \le k$$
$$1 \le j_1 \le deg(R_{i_1}), \ldots, 1 \le j_r \le deg(R_{i_r})$$

where $deg(R)$ is the *degree* of R, that is, the number of attributes in each tuple of R. The corresponding tuple calculus statement is

$$\Big\{ w^{(r)} \mid (\exists t_1) \cdots (\exists t_k)$$
$$(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$$
$$\wedge\, w[1] = t_{i_1}[j_1] \wedge \cdots \wedge w[r] = t_{i_r}[j_r]$$
$$\wedge\, \psi') \Big\}$$

This statement specifies that the tuple t_i is in the relation R_i, the result tuple w is composed of r attributes, the m-th attribute of w is copied from the j_m-th attribute of the tuple variable t_{i_m}, and that the participating tuples are determined by the restriction ψ'. We use ψ' instead of ψ to indicate modifications for attribute names and Quel syntax conventions.

1.3. Adding Aggregates to Tuple Relational Calculus

The semantics for the Quel retrieve statement with aggregates will be presented now. We first introduce the *aggregate operators* to be used in the tuple calculus. This material is new, and is based on Klug's method, which was used in a separate, more formal tuple relational calculus [Klug 1982]. In this approach, *an aggregate operator, defined as a function, is applied to a set of r-tuples, resulting in a tuple containing r* attribute values, with each attribute value equivalent to applying the aggregate over that attribute. By applying the function to the set of complete tuples, the distinction between unique and non-unique aggregation can be preserved.

Let R be a relation of degree r containing n tuples, $n \geq 0$, and let t be a tuple variable associated with R.

DEFINITION. $count(R) \triangleq (n, ..., n)$

That is, the count function yields a tuple whose r components equal n.

DEFINITION. $any(R) \triangleq (sign(n), ..., sign(n))$

The sign function produces the value +1 if n is positive (at least one tuple in R), and 0 if n is zero (no tuples in R). Again, all r components of the result tuple equal the same value.

For the remaining definitions, assume $n > 0$.

DEFINITION. $sum(R) \triangleq \left[\sum_{t \in R} t[1], ..., \sum_{t \in R} t[r] \right]$

Each component of the result tuple equals the sum of all values in the corresponding component of the tuples of R.

DEFINITION. $avg(R) \triangleq \left[\frac{1}{n}\sum_{t \in R} t[1], \ldots, \frac{1}{n}\sum_{t \in R} t[r]\right]$

Each component of the result tuple equals the average or arithmetic mean of all values in the corresponding component of the tuples of R.

DEFINITION. $min(R) \triangleq (\min_{t \in R} t[1], \ldots, \min_{t \in R} t[r])$

Each component of the result tuple equals the minimum of all values in the corresponding component of the tuples of R.

DEFINITION. $max(R) \triangleq (\max_{t \in R} t[1], \ldots, \max_{t \in R} t[r])$

Each component of the result tuple equals the maximum of all values in the corresponding component of the tuples of R.

For $n = 0$, *sum*, *avg*, *min* and *max* are arbitrarily defined to be 0. However, new implementations can be more consistent with reality if they return a special null value for those cases [Epstein 1979].

The advantage of defining aggregate operators to work on relations instead of on domains is that duplicate values enter the set calculations without difficulty. Later on we consider unique aggregates, which eliminate duplicate values to compute aggregates over unique values.

The functions defined above are used in the tuple calculus semantics. Let F be any of the aggregates defined in Section 1.1. Quel queries with one aggregate function in the target list are of the form

range of t_1 **is** R_1
...
range of t_k **is** R_k
retrieve $(t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r}, y = F(t_{l_1}.D_{m_1}$ **by** $t_{l_2}.D_{m_2}, \ldots, t_{l_n}.D_{m_n}$ **where** $\psi_1))$
    where ψ

in which

$$\begin{array}{l} 1 \le i_1 \le k, \ldots, 1 \le i_r \le k \\ 1 \le l_1 \le k, \ldots, 1 \le l_n \le k \\ 1 \le j_1 \le deg(R_{i_1}), \ldots, 1 \le j_r \le deg(R_{i_r}) \\ 1 \le m_1 \le deg(R_{l_1}), \ldots, 1 \le m_n \le deg(R_{l_n}). \end{array}$$

Again, we simplify the expressions appearing in the aggregate to attribute names. There is also the restriction that the tuple variable(s) mentioned in ψ_1 must be either t_{l_1} or one of the tuple variables appearing in the by clause: $t_{l_2}, \ldots, t_{l_n}$ (otherwise, there may be many more tuples participating in the aggregate, i.e., those from additional tuple variables, thereby generating unexpected results from the aggregate). The attributes outside the aggregate, $D_{j_1}, \cdots, D_{j_r}$, and the attributes used within the aggregate, $D_{m_1}, \cdots, D_{m_n}$, usually overlap, but need not. This aggregate

(a) takes the cartesian product of the relations associated with the tuple variables appearing in the aggregate,

(b) removes all resulting tuples that do not satisfy the condition in the where clause of the aggregate,

(c) partitions the resulting tuples by the values of the attributes listed in the by clause,

(d) applies the aggregate to each partition,

(e) and finally associates the result with each combination of tuples participating in the original query, with the partition selected using the values indicated in the by clause.

We first specify the partition of the cartesian product of the relations associated with the tuple variables appearing in the aggregate. Initially assume that the tuple variables $t_{l_1}, \ldots, t_{l_n}$ are all distinct. Define a *partitioning function* P corresponding to the aggregate in the query as a function of $n-1$ values $a_2, \ldots, a_n$, given by

$$
\begin{aligned}
P(a_2, \ldots, a_n) \triangleq \Big\{ b^{(p)} \mid (\exists t_{l_1}) \cdots (\exists t_{l_n}) \\
&(R_{l_1}(t_{l_1}) \wedge \cdots \wedge R_{l_n}(t_{l_n}) \\
&\wedge b[1] = t_{l_1}[1] \wedge \cdots \wedge b[p] = t_{l_n}[deg(R_{l_n})] \\
&\wedge t_{l_2}[m_2] = a_2 \wedge \cdots \wedge t_{l_n}[m_n] = a_n \\
&\wedge \psi_1) \Big\}
\end{aligned}
$$

where $p \triangleq \sum_{i=1}^{n} deg(R_{l_i})$. Each of the combinations of values $a_2, \ldots, a_n$ existing in the specified attributes produces one partition on which the aggregate has to be applied.

EXAMPLE. The partitioning function for Example 1 is particularly simple:

$$P(a_2) = \left\{ b^{(3)} \mid (\exists f)(Faculty(f) \wedge b = f \wedge f[Rank] = a_2) \right\}$$

For this particular *Faculty* relation, P(Assistant) = {(Tom, Assistant, 23000), (Merrie, Assistant, 25000)} and P(Associate) = {(Jane, Associate, 33000)}. Note that we use attribute names rather than indices for notational convenience. ||||||

Let F be the aggregate operator defined above corresponding to the Quel aggregate `F` (e.g., if `F` is `count`, F is *count*). A term of the form $F(R)$ will denote the tuple obtained from the application of aggregate operator F to relation R. The operator F applies the same aggregate to every attribute in R. Let $F(P(a_2, ..., a_n))[m]$ denote the m-th attribute of the tuple evaluated by $F(P(a_2, ..., a_n))$. For Example 1, $count(P(\text{Assistant})) = \{(2, 2, 2)\}$ and $count(P(\text{Assistant}))[Name] = 2$.

The counterpart tuple calculus statement for the Quel query is then

$$\Big\{ w^{(r+1)} \mid (\exists t_1) \cdots (\exists t_k)$$
$$(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$$
$$\wedge\, w[1] = t_{i_1}[j_1] \wedge \cdots \wedge w[r] = t_{i_r}[j_r]$$
$$\wedge\, w[r+1] = F(P(t_{l_1}[m_2], ..., t_{l_n}[m_n]))[m_1]$$
$$\wedge\, \psi') \Big\}$$

The partitioning function computes the partitions, with the appropriate partition selected by the parameter(s) passed to P. If the tuple variables appearing in the aggregate are not distinct, then the first two lines in the definition of P should be altered to eliminate duplicate tuple variables. Also, if tuple variable t_{l_1} does not appear outside of the aggregate or in the by clause, then that tuple variable should be removed from the first two lines.

EXAMPLE. The tuple calculus statement for Example 1 is

$$\Big\{ w^{(2)} \mid (\exists f)(Faculty(f) \wedge w[1] = f[Rank] \wedge w[2] = count(P(f[Rank]))[Name]) \Big\} \quad ||||||$$

For a scalar aggregate, there is no by clause and the partitioning function P is a set rather than a function, namely

$$P \triangleq \Big\{ b^{(p)} \mid (\exists t_{l_1})(R_{l_1}(t_{l_1}) \wedge b = t_{l_1} \wedge \psi_1') \Big\}$$

where $p = deg(R_{l_1})$. Here, P is formulated to emphasize its similarity with the more general partitioning function given earlier. As expected, P computes a subset of R_{l_1}. The tuple calculus statement for the query remains the same as above, except that P is used in place of $P\ (t_{l_1}[m_2], ..., t_{l_n}[m_n])$.

EXAMPLE. For the `count` aggregate of Example 2,

$$P_1 = \left\{ b^{(3)} \mid (\exists f)(Faculty(f) \wedge b = f\ ) \right\}$$ |||||||

For a query involving several aggregates, a separate partitioning function P (of either the scalar or functional form) is defined for each aggregate.

1.4. Unique Aggregation

The aggregates as defined cannot do unique aggregation directly, because they operate on relations, not on attributes. It turns out, however, that a slight change of the partitioning function P solves the problem.

Let the modified partitioning function be defined in terms of P as

$$U\ (a_2, ..., a_n) \triangleq \left\{ u^{(1)} \mid (\exists b)(b \in P\ (a_2, ..., a_n) \wedge u[1] = b[m_1]) \right\}$$

The net effect of this is the elimination of all duplicate values from the attribute upon which aggregation will be performed.

For a scalar unique aggregate, the partitioning *set* U is defined in a similar fashion based on P,

$$U \triangleq \left\{ u^{(1)} \mid (\exists b)(b \in P \wedge u[1] = b[m_1]) \right\}$$

The tuple calculus semantics of all unique aggregates is simply obtained by substituting U for P in the main formula of the previous section, and using the previously defined operators *count*, *sum*, and *avg*.

EXAMPLE. For Example 2 for the `countU` aggregate,

$$P_2 = \left\{ b^{(3)} \mid (\exists f)(Faculty(f) \wedge b = f\ ) \right\}$$

$$U_2 = \left\{ u^{(1)} \mid (\exists b)(b \in P_2 \wedge u[1] = b[2]) \right\}$$

$$= \{(\text{Assistant}), (\text{Associate})\}$$ |||||||

1.5. Multiple Aggregation

A Quel query may contain multiple aggregates. Each of the aggregates is computed from its own partitioning function. All the partitioning functions are then referenced in the main tuple calculus statement.

EXAMPLE. The query in Example 2 contains both `count` and `countU` aggregates. We gave the definitions for the two partitioning functions (actually sets) P_1 and U_2 above. The corresponding tuple tuple calculus expression is then

$$\left\{ w^{(2)} \mid w[1] = count(P_1)[Name] \wedge w[2] = count(U_2)[Rank]\,) \right\}$$

Since the tuple variable `f` being aggregated over does not appear outside of the aggregate, it also does not appear in the tuple calculus statement. ||||||

1.6. Aggregates in the Outer Where Clause

So far we have seen standard and unique aggregates being used in the target list of a query. They can also appear in the Quel where clause.

Let us first deal with an aggregate in the main where clause. If it is a scalar aggregate, it is independent of the rest of the query and therefore it is simply calculated and replaced by its value. However, if an aggregate function appears in the outer where clause, its corresponding partitioning function is defined, and the values of the aggregated attribute are used in place of the aggregate in the query. Following the rule that the tuple variables in by-lists are global, the by clause is linked to the rest of the query through the arguments to the partitioning function.

1.7. Nested Aggregation

A similar rule applies in the case of nested aggregation, that is, when an aggregate function F_3 appears in a local where clause of an aggregate F_2. The tuple variables in the by-list of F_3 are linked to the tuple variables of the same name appearing in their outer environment (that is, the F_2 query).

Nesting may be deeper, with F_2 nested in (called from) an outer aggregate F_1. Again, tuple variables appearing in the by-list of F_2 are linked to the tuple variables of the same name appearing in F_1, and so on.

Links are accomplished via the arguments to the partitioning functions. Thus, at any one time, only one level of nesting need be considered [Epstein 1979].

1.8. Expressions in Aggregates

In the formal semantics, we assumed that a single attribute was aggregated, after partitioning by zero or more attribute values. Quel allows arbitrary expressions to be aggregated, and supports expressions in the by clause. The former can be accommodated by simply substituting the appropriate expression for $F(\cdots)$ in the line specifying the output aggregate attribute in the main tuple calculus statement.

EXAMPLE. If Example 1 was modified to

```
range of f is Faculty
retrieve (f.Rank, This=count(f.Name by f.Rank)*count(f.Salary by f.Rank))
```

Example 3: One modification of Example 1.

the only change would be in the computation of $w[2]$:

$$w[2] = count(P(f[Rank]))[Name]*count(P(f[Rank]))[Salary] \quad ||||||$$

Expressions in the by clause require two changes: one in the definition of the partitioning function where the parameters are equated and one in the main statement, where values of the parameters are specified.

EXAMPLE. If Example 1 was modified to

```
range of f is Faculty
retrieve (f.Rank, This = count(f.Name by f.Salary mod 1000))
```

Example 4: Another modification of Example 1.

the modified partioning function definition and tuple calculus statement would be

$$P(a_2) = \left\{ b^{(2)} \mid (\exists f)(Faculty(f) \wedge b = f \wedge f[Salary] \; mod \; 1000 = a_2) \right\}$$

$$\left\{ w^{(2)} \mid (\exists f)(Faculty(f) \wedge w[1] = f[Rank] \wedge w[2] = count(P(f[Salary] mod\, 1000))[Name]) \right\} \quad ||||||$$

1.9. Summary

There are six fundamental operators that perform aggregation in Quel. The grouping and selection of tuples to be aggregated is done by the partitioning function, which also determines whether the standard or the unique version is being used. Aggregates may appear in the outer where clause, as well as nested in the inner where clause. The depth of nesting is arbitrary.

While only the semantics for the retrieve statement has been given, it is easy to extend it to specify aggregates in the Quel modification statements (**`append`**, **`delete`**, and **`replace`**) [Snodgrass 1987], using the strategy discussed in this section.

2. Temporal Aggregates In TQuel

In the previous section we have seen the various Quel aggregates and their formal semantics. We now introduce TQuel aggregates in an intuitive way through examples. We first give an overview of the TQuel language and then turn to aggregates.

TQuel is a version of Quel, augmented to handle the time dimension [Snodgrass 1987]. TQuel supports valid, transaction, and user-defined time, and thus supports temporal queries [Snodgrass & Ahn 1986]. Of the three, valid time, modeling the real world occurrence of an event, is by far the hardest to support in aggregates. Transaction time, modeling the storage of information in a database, may be supported through one additional term in the tuple calculus semantics. User-defined time, an encoding whose semantics is maintained by application programs, is handled in an identical manner to more conventional data types such as integers and character strings; all that is necessary are input, output, and comparison functions. To simplify the exposition, we will not use transaction or user-defined time in the example queries or in their formal semantics. In the general formal semantics, we will include transaction time, to illustrate how easy it is to support.

Temporal relations are four dimensional. Multiple tuples containing multiple attribute values contribute two dimensions; valid and transaction time contribute the other two dimensions. For both the examples and the semantics, we embed these four dimensional structures into two dimensional tables, appending additional, implicit time attributes that are not directly accessible to the user. Other embeddings are possible (five are given in [Snodgrass 1987]), but will not be used here. The degree (*deg*) of a temporal relation is the number of explicit attributes.

Relations in TQuel can represent either a collection of events that happen *at* certain points in time (event relations), or a collection of intervals that have a duration, that is, a *from* time and a *to* time (interval relations). Thus, event relations have one valid-time attribute, *at*, whose value represents an interval of unit duration, whose length depends on the *granularity* of valid time. In the examples, we have assumed a timestamp granularity of one month: events occurring within a month cannot be distinguished in time. Interval relations have two valid-time attributes, *from* and *to*, whose values together represent an interval of arbitrary length. t_1, when assigned to the valid-time attribute *at*, represents the interval $[t_1, t_1+1)$. If t_1 simply precedes t_2 in the linear ordering of time, then t_1 and t_2, when assigned to the valid-time attributes *from* and *to* respectively, represent the interval $[t_1, t_2)$. Although not shown in the examples, both event and interval relations carry two transaction-time attributes, *start* and *stop*, indicating when the tuple was recorded in the database and when it was logically deleted from the database, respectively. The assignment of the transaction times to a target relation is made by the system when data are recorded.

The TQuel retrieve statement augments the standard Quel retrieve statement by including

- a *when* clause, paralleling the already existing where clause, to select tuples whose temporal attributes satisfy desired temporal constraints;
- a *valid-at* clause that permits the assignment of a non-default and possibly computed value to the valid-time attribute of a target event relation;
- *valid-from* and *valid-to* clauses that permit the same kind of assignment to the valid-time attributes of a target interval relation; and
- an *as-of* clause to specify rollback to a previous transaction or series of transactions.

EXAMPLE. The relations *Faculty*, *Submitted* and *Published*, shorter versions of those appearing in [Snodgrass 1987], contain the following tuples:

Faculty(Name, Rank, Salary):

Name	*Rank*	*Salary*	*from*	*to*
Jane	Assistant	25000	9-71	12-76
Jane	Associate	33000	12-76	11-80
Jane	Full	34000	11-80	12-83
Jane	Full	44000	12-83	∞
Merrie	Assistant	25000	9-77	12-82
Merrie	Associate	40000	12-82	∞
Tom	Assistant	23000	9-75	12-80

Submitted(Author, Journal):

Author	*Journal*	*at*
Jane	CACM	11-79
Merrie	CACM	9-78
Merrie	TODS	5-79
Merrie	JACM	8-82

Published(Author, Journal):

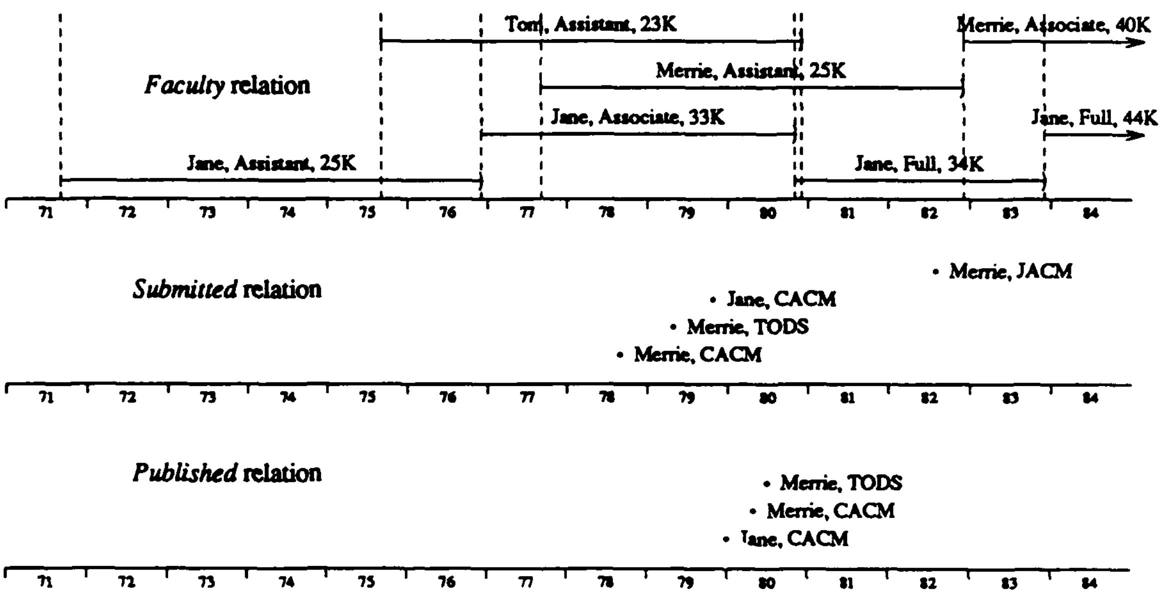
Author	*Journal*	*at*
Jane	CACM	1-80
Merrie	CACM	5-80
Merrie	TODS	7-80

A representation of the tuples in the three relations is shown in Figure 1. The first example TQuel query contains no aggregates:

```
range of f is Faculty
range of f2 is Faculty
retrieve (f.Rank)
valid at begin of f2
where f.Name = "Jane" and f2.Name = "Merrie" and f2.Rank = "Associate"
when f overlap begin of f2
```

Example 5: What was Jane's rank when Merrie was promoted to Associate?

Figure 1: Example Relations shown on a Time Line



Only two tuples will participate in this query, (Jane, Full, 34000, 11-80, 12-83) for f and (Merrie, Associate, 40000, 12-82, ∞) for f2, based on the where and when clauses. The target list specifies the value of the *Rank* attribute and the valid-at clause specifies the value of the implicit *at* attribute. The resulting relation has one tuple,

Rank	*at*
Full	12-82

2.1. Adding Aggregates to TQuel

It is desirable that TQuel aggregates be a superset of the Quel aggregates, with a natural time-oriented interpretation. Therefore, the TQuel version of a Quel aggregate will perform the same fundamental operation, while ranging over an event or an interval relation.

There are some differences between Quel and TQuel aggregates. Historical and temporal databases are characterized by the changing condition of their relations: at time t_1 a relation contains a set of tuples, and at time t_2 the same relation may contain a different set. Since aggregates are computed from the entire relation, this in turn causes the value of an aggregate to change from, say, v_1 to v_2. Hence, while in Quel an aggregate with no by-list (scalar aggregate) returns a *single* value, in TQuel the same aggregate returns, generally speaking, a *sequence* of values, each attached to its valid times. For an aggregate with a by-list, a sequence of values for each value in the by-list is generated.

EXAMPLE. Let us consider Example 1, this time on an historical relation:

```
range of f is Faculty
retrieve (f.Rank, NumInRank = count(f.Name by f.Rank))
```

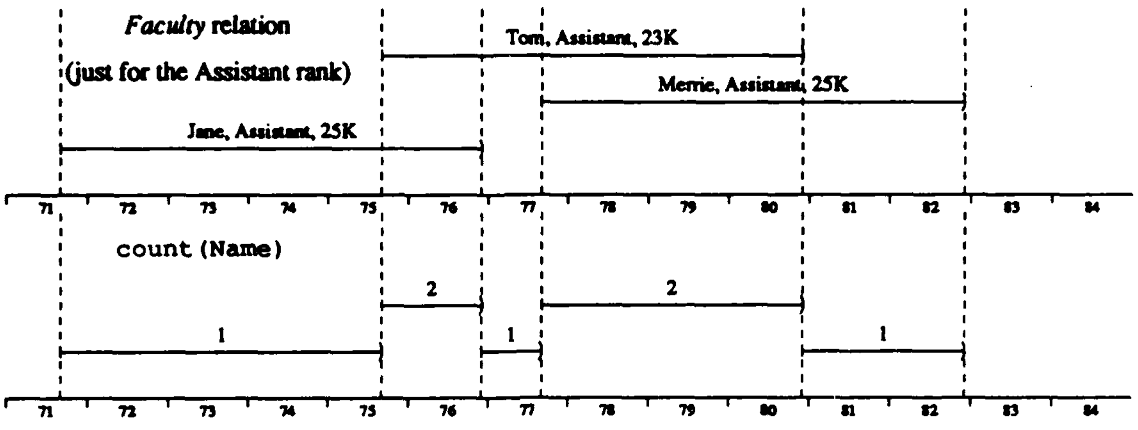
Example 6: Example 1 on an historical relation.

This query retrieves each rank, together with the current number of faculty at that rank. With the default when clause (**when** f **overlap now**) and valid clause (**valid from begin of** f **to end of** f), the resulting relation is

Rank	*NumInRank*	*from*	*to*
Associate	1	12-82	∞
Full	1	12-83	∞

Defaults are discussed in detail in Section 2.5. To extract the *history* of the requested count, simply use an explicit when clause: **when true**. As can be seen in Figure 2, for each rank there can be more than one related count over time.

Figure 2: An Example of count



The altered query yields the following tuples

Rank	*NumInRank*	*from*	*to*
Assistant	1	9-71	9-75
Assistant	2	9-75	12-76
Assistant	1	12-76	9-77
Assistant	2	9-77	12-80
Assistant	1	12-80	12-82
Associate	1	12-76	11-80
Associate	1	12-82	∞
Full	1	11-80	12-83
Full	1	12-83	∞

The count may change only when a Faculty tuple is created, or becomes invalid. Thus each output tuple is valid between two events (represented by vertical dotted lines) in the graph of the *Faculty* relation (Figure 1). ||||||

EXAMPLE. The next example shows how an aggregate, which gives an interval relation, can occur with an event relation in a query.

```
range of f is Faculty
range of s is Submitted
retrieve (s.Author, s.Journal, NumFac = count(f.Name))
when s overlap f
```

Example 7: How many faculty members were there each time a paper was submitted to a journal?

The result is:

Author	*Journal*	*NumFac*	*at*
Merrie	CACM	3	9-78
Merrie	TODS	3	5-79
Jane	CACM	3	11-79
Merrie	JACM	2	8-82

The count is computed for every period of time such that `f` overlaps `s`, and then, by default, the valid times of the output are the overlap of the valid times of the count, the `f` tuple variable, and the `s` tuple variable, producing an event relation. ||||||

Quel allows an inner where clause as the way to preselect tuples for the computation of the aggregate; otherwise, aggregates always operate on the entire relation. Similarly, in TQuel the inner where, when, and as-of clauses serve the same purpose. An inner valid clause is not allowed, because the interval of validity for the value calculated by the aggregate is indirectly specified using the for clause, to be discussed in Section 2.2.

EXAMPLE. Consider the query in Example 6, modified to exclude Jane from the calculation of the aggregate:

```
range of f is Faculty
retrieve (f.Rank, NumInRank=count(f.Name by f.Rank where f.Name!="Jane"))
```

Example 8: A third modification of Example 1

Again, with the default when and valid clauses, the query yields the following tuples

Rank	*NumInRank*	*from*	*to*
Associate	1	12-82	∞
Full	0	12-83	∞

Note that a default value of zero occurs for each point in time when a tuple of the specified rank is valid, but the subset of tuples used to compute the aggregate is empty. ||||||

The above examples illustrate our approach to computing TQuel aggregates. To aggregate a given attribute of relation R,

(a) Determine the periods of time during which R remained "fixed" or "constant", that is, no new tuples entered the relation (and, if R is an interval relation, no tuples became invalid).

(b) For each constant set of tuples in R, select the tuples that satisfy all the qualifications required by the inner where, when, and as-of clauses, if any. Defaults are used if those clauses are not present.

(c) If there is a by-list with this aggregate, subdivide each constant set of tuples into subsets, each subset corresponding to one value of the by-list attributes. Each group of selected tuples is called an *aggregation set.*

(d) Compute the aggregate for each aggregation set.

(e) Associate the result with each combination of tuples participating in the original query, with the aggregation set selected (1) using the values indicated in the by clause, (2) using the valid time of the tuple variables appearing in the aggregate, and (3) using the interval or event specified in the valid clause.

The basic strategy consists of reducing a TQuel aggregate to a series of Quel-style aggregates, each applied on a period of time when the relation does not change its contents. Each value of the aggregate is associated with an assignment of values to the by-list attributes, and is attached to the particular period of time it was valid. At each point in time, there is exactly one value of the aggregate for each combination of values of the by-list attributes.

This approach is necessarily more complex than that given in Section 1.3 for Quel aggregates. In TQuel, for each interval during which all base relations participating in the aggregate(s) remain "fixed," an aggregate tuple is computed for each aggregation set. In Quel, all base relations are already fixed, since the relations do not vary over time. This aggregate tuple, along with tuples from the base relations that are valid over the interval, determine the output tuples for the interval. Whereas Quel uses only the explicit attribute values via the by clause to connect the aggregate tuple with the participating tuples in the retrieve statement, TQuel also uses the implicit time values. Any combination of aggregate and base-relation tuples that satisfy all qualifications required by the outer where and when clauses, and also overlap, produce an output tuple. In addition, the valid time of each output tuple is required to be the overlap of the interval or event specified by the valid clause with the overlap of the aggregate tuple and base-relation tuples named in the aggregate.

The restriction that the valid time of the output tuple be the intersection of the valid times of some of the participating tuples and the aggregate tuple as well as the time specified by the valid clause does not limit the range of queries that TQuel can support. To support queries whose output is derived from aggregate and base-relation tuples valid over different intervals, we simply pre-compute the aggregates and treat them as ordinary historical relations in the main TQuel query.

EXAMPLE. The following query combines information from two separate intervals of time.

```
range of f is Faculty
retrieve into temp (maxsal = max(f.Salary))
range of t is temp
retrieve (f.Name)
valid at "June, 1981"
where f.Salary > t.maxsal
when f overlap "June, 1981" and t overlap "June, 1979"
```

Example 9: Who made a salary in June, 1981 that exceeded the maximum salary made in June, 1979?

With the default when clause (`when true`) and valid clause (`valid from begining to forever`) for the first retrieve statement, the query yields

Name	*at*
Jane	6-81

By pre-computing the aggregate and substituting the resulting historical relations for references to the aggregate in the main query, we have in effect reduced the TQuel query with aggregates to a TQuel query without aggregates. Hence, there are no implied restrictions on the valid times of the aggregate and base-relation tuples that contribute to output tuples or the valid time of the output tuples.

2.2. Cumulative versus Instantaneous Aggregates

An aggregate may or may not take into account tuples that are no longer valid. The following definitions are useful:

Cumulative Aggregates. If the value returned by an aggregate for each point *t* in time is computed from all tuples that have been valid in the past, as well as those valid at *t*, then the aggregate is said to be *cumulative.*

Instantaneous Aggregates. If the value returned by an aggregate for each point *t* in time is computed only from the tuples valid at time *t*, then the aggregate is said to be *instantaneous.*

These aggregates act differently when applied to an event or an interval relation. For an event relation, as the length of the time unit (the timestamp granularity) is reduced, the probability of finding any valid tuples decreases. Aggregates such as `count`, applied at a given instant, would thus return different results depending upon the granularity of valid time. On the other hand, it is always possible to count the events that have occurred in the past, or in a given period of time, in a cumulative fashion. For an interval relation, tuples are valid over an interval of time which is at least as long as the timestamp granularity, and therefore the above problem does not exist. We therefore restrict aggregate operators over event relations to be cumulative, while aggregate operators over interval relations can have both an instantaneous and a cumulative version. However, each value of an aggregate, be it instantaneous or cumulative, is valid during a period of time.

For cumulative aggregates, the user must specify how far in the past to include tuples used to compute a value at time t. The for clause is used for this purpose. Instantaneous aggregates (the default) are specified using **for each instant**. If all previous tuples are to participate, **for ever** is used. Intermediate cases, such as using only those tuples valid at some point in the previous year, are specified using **for each** < time unit>, e.g., **for each year**, **for each day**. If, say, count(**for each year**) is used, then the aggregate, when computing a value valid at a particular month m, will operate over all tuples that were valid sometime during the year up to and including the month m. The value at 3-76 will include all tuples valid sometime during 4-75 through 3-76; the value at 4-76 will include the (potentially different) tuples valid sometime during 5-75 through 4-76. The interval used (in this case, year) is termed the *window*, and such aggregates are termed *moving-window aggregates*.

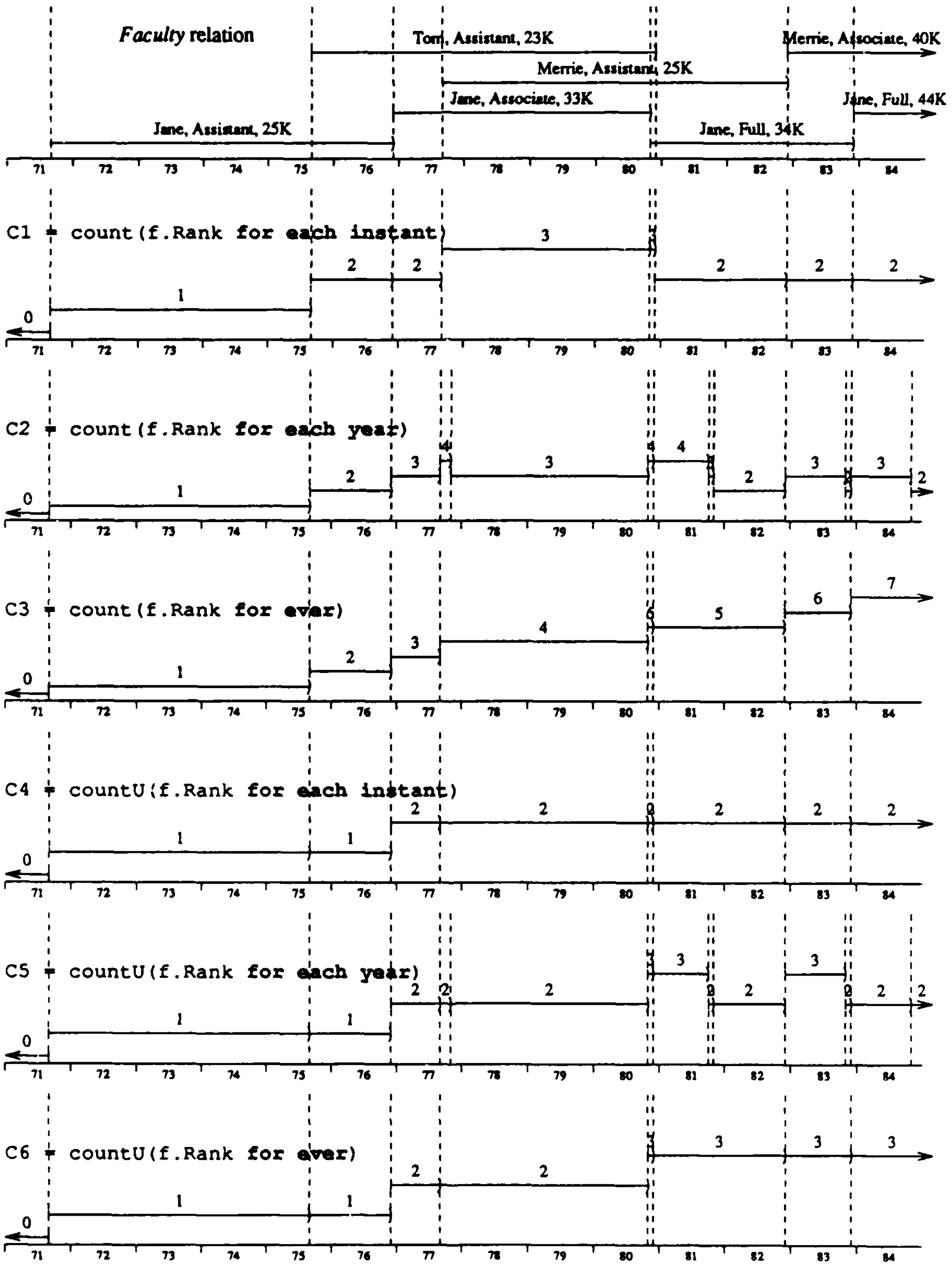
EXAMPLE. To illustrate the difference between the various kinds of aggregates of an interval relation, consider Figure 3, which illustrates the execution of the following query,

```
range of f is Faculty
retrieve (C1= count(f.Rank for each instant), C2=count(f.Rank for each year),
          C3=count(f.Rank for ever), C4=countU(f.Rank for each instant),
          C5=countU(f.Rank for each year), C6=countU(f.Rank for ever))
```

Example 10: Various combinations of unique and window sizes.

on the historical *Faculty* relation shown in Figure 1. Because the tuple variable f does not appear outside the aggregates, the default when clause is **when true**. Hence, the entire history of the counts is computed. ▮▮▮

Figure 3: Comparison of Six Aggregate Variants



Faculty relation
Tom, Assistant, 23K
Merrie, Associate, 40K
Merrie, Assistant, 25K
Jane, Associate, 33K
Jane, Full, 44K
Jane, Assistant, 25K
Jane, Full, 34K
71
72
73
74
75
76
77
78
79
80
81
82
83
84
C1 = count(f.Rank for each instant)
C2 = count(f.Rank for each year)
C3 = count(f.Rank for ever)
C4 = countU(f.Rank for each instant)
C5 = countU(f.Rank for each year)
C6 = countU(f.Rank for ever)

2.3. New Aggregates

All Quel aggregates have a TQuel counterpart. There are also some aggregates unique to TQuel. The first is quite similar to `avg`, applying both to snapshot relations and temporal relations:

`stdev` The standard deviation of the set of *n* values present in a given attribute, defined as a measure of the homogeneity of the values. This operator is restricted to operate only on numeric attributes.

The remaining new aggregates are strictly temporal.

`first` This aggregate returns, at each point in time, the oldest value of the given attribute, that is, the one associated with the first valid tuple. If two tuples have the same *from* value, one is arbitrarily selected.

`last` This aggregate is analogous to `first`. It returns, at each point in time, the newest value of the given attribute, that is, the one associated with the tuple with the latest *from* time. If two tuples have the same *from* time, one is arbitrarily selected.

`avgti` *AVeraGe Time Increment*: the average growth or decrease experienced by values of an attribute over time. This aggregate is only applicable to numeric attributes in event relations. It returns a value indicating growth per time unit, e.g., feet/hour, or dollars/month. The time unit can be optionally specified by the user by means of the **`per`** clause (see the syntax in the appendix): **`per hour, per month`**. This aggregate compares the attribute value of each tuple with the attribute value of its chronologically previous tuple, relative to the time elapsed, and smooths out all the comparisons by taking their arithmetic mean. At least two tuples are needed to compute `avgti` so that the comparison can be made; when there are less than two tuples, a value of 0 results.

`varts` *VARiability of Time Spacing*: the degree of inequality of the time spacing within a given set of events (the argument to this aggregate is an event expression evaluating to an event). This aggregate returns a nondimensional quantity which has the same value for each attribute. A value of 0 indicates the tuples are perfectly spaced. This aggregate also considers the tuples in chronological order. It finds the ratio of the standard deviation of the time lengths from one tuple to the next, to the average of those time lengths. Like in `avgti`, at least two tuples are needed to perform the comparison, with a 0 resulting when two tuples aren't available.

In addition, two aggregates that evaluate to valid time are available.

`earliest` The oldest time period of an interval relation, that is, the first *from-to* interval or the oldest event, that is, the first *at* event. If two tuples of an interval relation have the same *from* value, the one with the earlier *to* time is considered to be older.

`latest` The newest time period of an interval relation, that is, the last *from-to* interval or the newest event, that is, the last *at* event. If two tuples of an interval relation have the same *from* value, the one with the later *to* time is considered to be newer.

If no tuples are available to aggregate over (i.e., if the aggregation set is empty), then `first` and `last` return a distinguished value for each datatype (e.g., 0 for integer attributes), and `earliest` and `latest` return the interval **`beginning extend forever`**. They are called *aggregated temporal constructors* because they return a time interval as their result. They can be employed by the user to specify conditions in the temporal qualification (when clause) or the valid time (valid clause). To adhere to the syntax of temporal expressions and predicates, these aggregates take an interval expression, rather than an a numeric or string valued expression, as an argument.

Note that, while `first` and `last` yield (potentially) several tuples of output, `first(for ever)` outputs just one tuple. The same comment applies to `earliest(for ever)`.

2.4. More Examples

The next example, modified from one given in [Epstein 1979], shows an aggregate in the inner where clause of another aggregate; a case of nested aggregation:

```
range of f is Faculty
retrieve (f.Name, f.Salary)
valid from begin of f to begin of "1980"
where f.Salary = min(f.salary for each instant
                            where f.Salary != min(f.Salary for each instant))
when true
```

Example 11: Who was making the second smallest salary, and how much was it, during each period of time prior to 1980?

The output is

Name	*Salary*	*from*	*to*
Jane	25000	9-75	12-76
Jane	33000	12-76	9-77
Merrie	25000	9-77	1-80

Aggregates can also appear outside the target list:

```
range of f is Faculty
retrieve (f.Name, f.Rank)
when begin of earliest(f by f.Rank for ever) precede begin of f
    and begin of f precede end of earliest(f by f.Rank for ever)
```

Example 12: Who were the professors hired into or promoted to a rank while the first faculty member ever in that rank had not yet been promoted?

The two portions of the when clause specify (1) that `f` was hired into or promoted after the earliest faculty member, and (2) that the earliest faculty member had not been subsequently promoted before `f` was promoted into the rank.

First the `earliest` in each rank is computed,

Rank	*earliest(f)*	*from*	*to*
Assistant	[9-71, 12-76)	9-71	∞
Associate	[12-76, 11-80)	12-76	∞
Full	[11-80, 12-83)	11-80	∞

Only one tuple satisfies the when clause, and the output is

Name	*Rank*	*from*	*to*
Tom	Assistant	9-75	12-80

The when clause can be used inside an aggregate:

```
range of f is Faculty
retrieve (amountct=countU(f.Salary for ever when begin of f precede "1981"))
valid at now
```

Example 13: How many different salary amounts has the department paid its members since its creation until 1981?

Through the use of `countU`, each salary amount is counted only once for each period of time. The result is

amountct	*at*
4	*now*

Note that Merrie's initial salary of 25K is not counted, because is is identical to Jane's initial salary.

Our last examples reference the event historical relation *experiment*, containing the following tuples:

experiment(Yield):

Yield	*at*
178	9-81
179	11-81
183	1-82
184	2-82
188	4-82
188	6-82
190	8-82
191	10-82
194	12-82

```
range of x is experiment
retrieve (VarSpacing = varts(x for ever),
          GrowthPerYear = avgti(x.Yield per year for ever))
valid at x
when true
```

Example 14: Given the above set of experimental data, how equally spaced are the observations in time, and how fast is the yield growing per year?

Since we want the history, we override the default when clause. Computation of the variability of time spacing, for any attribute, consists of (a) sorting tuples by their *at* attribute and (b) considering every pair of chronologically consecutive tuples, S_i and S_{i+1}, and finding the coefficient of variation of the length of time from event S_i to event S_{i+1}, that is,

$$\frac{\text{standard deviation of } \langle S_2[at] - S_1[at], \cdots, S_{i+1}[at] - S_i[at] \rangle}{\text{average of } \langle S_2[at] - S_1[at], \cdots, S_{i+1}[at] - S_i[at] \rangle}$$

To compute the average time increment, we (a) again sort the tuples by their *at* attribute, and (b) for each pair of chronologically consecutive tuples S_i and S_{i+1}, compute the increment of the value $S_{i+1}[Yield] - S_i[Yield]$, averaged over previous pairs (**for ever** implies over all previous pairs), and then normalize over a year (**per year**). The result is the following relation:

VarSpacing	*GrowthPerYear*	*at*
0.0000	0	9-81
0.0000	6	11-81
0.0000	15	1-82
0.2828	14	2-82
0.2474	16.5	4-82
0.2222	13.2	6-82
0.2033	13	8-82
0.1884	12	10-82
0.1764	12.8	12-82

The value of *VarSpacing* at 2-82 is fairly large because the previous four tuples (at 9-81, 11-81, 1-82, and 2-82) were quite variably spaced (2 months, 2 months, and 1 month). After that point, *VarSpacing* decreases with time. Since *VarSpacing* = 0 means that all tuples are equally time-spaced, the gradual decrease in *VarSpacing* means that the observations, as time passes, are approaching uniformity in their time spacing. Because of the number of elements required to compute a standard deviation, *VarSpacing* has a value of 0 before 2-82.

The *GrowthPerYear* at 11-81 results from an increase of 1 over two months, implying a yearly increase of 6. The value jumps to 15 at 1-82 due to the increment of 4 over the previous two months (an instantaneous growth of 24 per year). It then generally decreases with time, indicating that yearly yield is growing more slowly.

Sometimes the result is desired only at certain times, such as the end of the year. If relations such as

yearmarker(YearNumber):

Year	*from*	*to*
.	.	.
1970	1-70	1-71
1971	1-71	1-72
1972	1-72	1-73
.	.	.

are provided, then the following is possible:

```
range of x is experiment
range of y is yearmarker
retrieve (VarSpacing = varts(x for ever),
          GrowthPerYear = avgti(x.Yield per year for ever))
valid at end of y
when true
```

Example 15: A modification of Example 14.

resulting in the following relation:

VarSpacing	*GrowthPerYear*	*at*
0.0000	6	12-81
0.1764	12.8	12-82

If an analogous *monthmarker* relation is available, then the following statements

```
range of x is experiment
range of m is monthmarker
retrieve (VarSpacing = varts(x for ever),
          GrowthPerYear = avgti(x.Yield per year for ever))
valid at end of m
where m.MonthNumber mod 3 = 0
when true
```

Example 16: Example 15 on a quarterly basis.

result in the following relation:

VarSpacing	*GrowthPerYear*	*at*
0.0000	0	9-81
0.0000	6	12-81
0.2828	14	3-82
0.2222	13.2	6-82
0.2033	13	9-82
0.1764	12.8	12-82

2.5. Defaults

Defaults must be chosen carefully to maintain the snapshot reducibility to Quel, thereby allowing TQuel aggregates to be used in exactly the same way as Quel aggregates. Each default may be overridden with the explicit use of the clause. There are two places where default clauses may apply: the outer retrieve statement and within the aggregate. The defaults clauses in the outer retrieve statement without aggregates was given in [Snodgrass 1987]:

```
valid from begin of (t₁ overlap ··· overlap tₖ) to end of (t₁ overlap ··· overlap tₖ)
where true
when t₁ overlap ··· overlap tₖ
as of now
```

where t_1, ..., t_k are the tuple variables appearing in the query.

When aggregates are included in the query, we must distinguish between the tuple variables appearing inside and outside the aggregate. Tuple variables are included in the default when and valid clauses *only* if they appear outside an aggregate. If no tuple variable appears outside an aggregate, the defaults are

```
valid from beginning to forever
where true
when true
as of now
```

The following defaults are assumed within each aggregate, and are quite similar to the defaults used in the outer query.

```
for each instant
where true
when t₁ overlap ··· overlap tₖ
as of α through β
```

where t_1, ..., t_k are the tuple variables appearing in the aggregate, and α and β are the expressions (or their defaults) appearing in the retrieval statement itself.

3. Tuple Calculus Semantics Of TQuel Aggregates

It is convenient to base the semantics of TQuel on the conventional (snapshot) relational database model, especially because of the available mathematical foundation supporting the latter [Codd 1972]. Thus the semantics of the augmented operations are expressed using traditional tuple calculus notation.

We first review the transformation of the time-specific constructs of TQuel into the tuple calculus, and briefly give the semantics of the TQuel retrieve statement, which is needed in order to introduce the semantics of temporal aggregates. This review is a condensation of [Snodgrass 1987]. The semantics of the TQuel aggregates is then developed.

3.1. Review of TQuel Semantics

As stated in the overview of TQuel in Section 2, TQuel augments Quel by adding a valid clause to specify the validity time(s) of tuples, a when clause to specify the relative time ordering of the participating tuples, and an as-of clause to specify rollback in time.

The semantics makes use of several auxiliary functions: temporal constructor functions that take one or two intervals and compute an interval, and temporal predicate functions (including *overlap*) that take two intervals and compute a boolean value. All of them are ultimately defined in terms of the predicates *Before* and *Equal* and two functions *first* and *last*.

The temporal predicate τ in the when clause, containing the `precede`, `overlap`, `and`, `or`, and `not` operations, is transformed into a standard tuple calculus predicate Γ_τ containing only the *Before*, *Equal*, $\wedge$, $\vee$, and $\neg$ operations. The valid clause is transformed into the functions Φ_υ and Φ_χ, each evaluating to an event, and containing the functions *first* and *last*. The as-of clause is in fact a special when clause stating that the transaction times of the underlying tuples must overlap the (constant) interval specified in the as-of clause. The constants Φ_α and Φ_β represent the endpoints of this interval from the expressions α and β. As a consequence, the query

```
range of $t_1$ is $R_1$
...
range of $t_k$ is $R_k$
retrieve ($t_{i_1}.D_{j_1}$, $\cdots$ , $t_{i_r}.D_{j_r}$)
    valid from $\upsilon$ to $\chi$
    where $\psi$
    when $\tau$
    as of $\alpha$ through $\beta$
```

is translated into the tuple calculus statement

$$
\Big\{ w^{(r+4)} \mid (\exists t_1) \cdots (\exists t_k) \Big(
(R_1(t_1) \land \cdots \land R_k(t_k))
\land w[1] = t_{i_1}[j_1] \land \cdots \land w[r] = t_{i_r}[j_r]
\land w[r+1] = \Phi_\upsilon \land w[r+2] = \Phi_\chi \land \mathit{Before}(w[r+1], w[r+2])
\land w[r+3] = \mathit{current\ transaction\ time} \land w[r+4] = \infty
\land \psi'
\land \Gamma_\tau
\land (\forall l)(1 \le l \le k)(\mathit{overlap}([\Phi_\alpha, \Phi_\beta), [t_l[\mathit{start}], t_l[\mathit{stop}])))
\Big) \Big\}
$$

The superscript indicates that the tuple w has r explicit attributes and 4 implicit attributes, indicating an interval relation. The semantics for an event relation is similar, but with only 3 implicit attributes, since the *to* time is not present.

EXAMPLE. Example 5, which results in an event relation, has the following tuple calculus semantics, ignoring transaction time.

$$
\begin{aligned}
\Big\{ w^{(l+1)} \mid\ & (\exists f)(\exists f2) \\
& (Faculty(f) \wedge Faculty(f2) \\
& \wedge w[1] = f[Rank] \\
& \wedge w[l+1] = f2[from] \\
& \wedge f[Name] = \texttt{"Jane"} \wedge f2[Name] = \texttt{"Merrie"} \wedge f2[Rank] = \texttt{"Associate"} \\
& \wedge overlap([f[from], f[to]), f2[from]) \\
&) \Big\} \quad ∎
\end{aligned}
$$

3.2. New TQuel Aggregates

Let us specify the semantics of the new aggregates introduced in Section 2.3. Let R be an event relation of degree r (recall that the degree only concerns the explicit attributes) with n tuples, $n > 2$. These aggregates all compute a single snapshot tuple of degree r.

DEFINITION.

$$
\begin{aligned}
S \triangleq chronorder(R) \Longleftrightarrow (\forall i)(1 \le i \le |S|)\ ((\exists t)\ (R(t) \wedge t = S_i))& \\
\wedge\ Before\ (S_{i-1}[at], S_i[at])& \\
\wedge\ S_{i-1}[at] \ne S_i[at]\ )&
\end{aligned}
$$

where $|S|$ is the length of the sequence S. Each element of S is a full tuple from R, and the elements of S are ordered by the *at* times of R. If several tuples in R show identical *at* times, only one of them is taken into S. Hence, the length of S is less than or equal to n.

DEFINITION. $avgti(R) \triangleq \left[\left[\dfrac{1}{|S|-1} \displaystyle\sum_{i=1}^{|S|-1} \dfrac{S_{i+1}[1] - S_i[1]}{S_{i+1}[at] - S_i[at]} \right], \ldots, \left[\dfrac{1}{|S|-1} \displaystyle\sum_{i=1}^{|S|-1} \dfrac{S_{i+1}[r] - S_i[r]}{S_{i+1}[at] - S_i[at]} \right] \right]$

where $S = chronorder(R)$ and $|S| > 1$. Each attribute of the result tuple equals the average increment (positive or negative) in the values of the corresponding attribute in R, per unit of time (the default is the timestamp granularity, defined in Section 2). An optional per clause can be used to specify the time unit desired; this causes multiplication of the result by a fixed conversion factor. For example, if timestamp granularity was a millisecond and the user specified "`per month`" then the computed result is multiplied by the conversion factor of milliseconds to months (2.592×10^9) before being output.

DEFINITION. $varts(R) \triangleq \dfrac{sd(D(R))}{mean(D(R))}$

where $D(R) \triangleq \langle d_1, \cdots, d_{|S|-1} \rangle$ such that $S = chronorder(R), |S| > 1$, implies that $(\exists i)\ (1 \le i \le |S|-1 \wedge d_i = S_{i+1}[at] - S_i[at])$, and $mean(X)$ and $sd(X)$ respectively denote the arithmetic mean and the arithmetic standard deviation of the real numbers in the set X. Each attribute of the result tuple equals the variability of the spacing between the *at* times among the tuples in R. This is in fact the coefficient of variation of the set $D(R)$. Note that *varts* returns a single value, rather than a tuple.

Observe that $mean(D(R))$ is never zero since $S_i[at]$ and $S_{i+1}[at]$ are distinct. Not necessarily all tuples from R will make their way into S; S was so defined in order to ensure that *avgti* or *varts* will not attempt a division by zero. Should the user need to specify which of the tuples from R has to be chosen for the chronological order, one of the other aggregates can be used to create a temporary relation T that contains the relevant tuples, and then `avgti` or `varts` may be applied to T.

Let R be an interval relation of degree r, and t be a tuple variable associated with R.

DEFINITION.

$$stdev(R) \triangleq \left[\sqrt{\frac{1}{n}\sum_{t\in R}(t[1])^2 - \frac{1}{n^2}(\sum_{t\in R} t[1])^2}, \ldots, \sqrt{\frac{1}{n}\sum_{t\in R}(t[r])^2 - \frac{1}{n^2}(\sum_{t\in R} t[r])^2}\right]$$

Each component of the result tuple equals the standard deviation of all values in the corresponding component of the tuples of R.

DEFINITION. $firstagg(R) \triangleq t_{first}$ where t_{first} satisfies the predicate

$$R(t_{first}) \wedge (\forall t)\,(R(t) \wedge t \neq t_{first} \Longrightarrow Before\,(t_{first}[r+1], t[r+1]) \vee Equal\,(t_{first}[r+1], t[r+1]))$$

The result tuple is the tuple whose valid times contain the earliest beginning time of a tuple in R, more specifically, no other tuple in R began before t_{first}. If R is empty, $t_{first} = (0, ..., 0, 0, \infty)$.

DEFINITION. $lastagg(R) \triangleq t_{last}$ where t_{last} satisfies the predicate

$$R(t_{last}) \wedge (\forall t)\,(R(t) \wedge t \neq t_{last} \Longrightarrow Before\,(t[r+1], t_{last}[r+1]) \vee Equal\,(t[r+1], t_{last}[r+1]))$$

The result tuple is the tuple whose valid times contain the latest beginning time of a tuple in R, more specifically, no other tuple in R began after t_{last}. If R is empty, $t_{last} = (0, ..., 0, 0, \infty)$.

The functions *firstagg* and *lastagg* directly support the aggregates `first` and `last`, respectively.

DEFINITION. $earliest(R) \triangleq [t_{earliest}[from], t_{earliest}[to])$ where $t_{earliest}$ satisfies the predicate

$$\begin{aligned} R(t_{earliest}) \wedge (\forall t)\,(R(t) \wedge t \neq t_{earliest} \Longrightarrow\ & Before\,(t_{earliest}[r+1], t[r+1]) \\ & \vee (Equal\,(t_{earliest}[r+1], t[r+1]) \wedge (Before\,(t_{earliest}[r+2], t[r+2]) \\ & \vee Equal\,(t_{earliest}[r+2], t[r+2])))) \end{aligned}$$

The result is the interval represented by the valid times of the earliest tuple in the relation.

DEFINITION. $latest(R) \triangleq [t_{latest}[from], t_{latest}[to])$ where t_{latest} satisfies the predicate

$$\begin{aligned} R(t_{latest}) \wedge (\forall t)\,(R(t) \wedge t \neq t_{latest} \Longrightarrow\ & Before\,(t[r+1], t_{latest}[r+1]) \\ & \vee (Equal\,(t[r+1], t_{latest}[r+1]) \wedge (Before\,(t[r+2], t_{latest}[r+2]) \\ & \vee Equal\,(t[r+2], t_{latest}[r+2])))) \end{aligned}$$

The result is the interval represented by the valid times of the latest tuple in the relation.

3.3. The Constant Predicate

As we have seen, aggregates change their values over time. This will be reflected as different values of an aggregate being associated with different valid times, even in queries that may look similar to Quel queries with scalar aggregates, in which no inner when or as-of clauses exist (recall the default clauses from Section 2.5). In TQuel, the role of the external or outer ***where***, ***when*** and ***as of*** clauses will be similar to that of the outer ***where*** in Quel: they determine which tuples from the underlying relations participate in the remainder of the query. These selected tuples are combined with the tuples computed from the aggregation sets to obtain the final output relation.

Aggregates always generate temporary interval relations, even though an aggregated attribute can appear in an event relation. The interval relation h.s exactly one value at any point in time (for an aggregate function, the interval relation has at most one value at any point in time for each value of attributes in the by list). It is convenient to determine the points at which the value changes. Let us first define the *time-partition* of a set of relations as

$$T(R_1, \ldots, R_k, w) \triangleq \left\{ s \mid (\exists r)(\exists i)(\exists t) \right.$$
$$(1 \le i \le k \wedge R_i(r) \wedge (s = r[from] \vee s = r[to] \vee s = t)$$
$$\left. \wedge\, t - w(t) = r[to] \;\wedge\; \forall t', t' > t, t' - w(t') > r[to]) \right\} \cup \left\{ 0, \infty \right\}$$

where w is an arbitrary function that maps each time t into its aggregation window size, with the single restriction that $w(t+1) \le w(t)+1$. The time-partition brings together all the times s when an aggregate in which the relations $R_1, \ldots, R_k$, mentioned in an aggregate, could change value. These times include the beginning time of each tuple, the time following the ending time of each tuple, and the time when a tuple no longer falls into an aggregation window. The window function w is specified in the for clause. `for each instant` implies $\forall t, w(t) = 0$; `for ever` implies $\forall t, w(t) = \infty$; and `for each` < time unit> implies a window size dependent on the timestamp granularity. In the examples, a granularity of month has been used. Hence, `for each month` is equivalent to `for each instant` ($\forall t, w(t) = 1-1 = 0$); `for each quarter` implies $\forall t, w(t) = 3-1 = 2$; and `for each decade` implies $\forall$

t, $w(t) = 120-1 = 119$. One is subtracted because the window is inclusive (see Section 2.2). If, however, a granularity of day is used, **for each month**, **for each quarter**, and **for each decade** would require non-constant window functions. For example, **for each month** would require w(January 31, 1980) = 30 and w(February 28, 1980) = 27.

If two times c and d are neighbors, i.e., in $T(R_1, ..., R_k, w)$, the time interval from c to d did not witness any change in the set of relations, or in other words, all the relations remained "constant". Define then the *Constant* predicate as

$$\begin{aligned}
Constant(R_1, ..., R_k, c, d, w) \Longleftrightarrow\ & c \in T(R_1, ..., R_k, w) \\
& \wedge d \in T(R_1, ..., R_k, w) \\
& \wedge c \neq d \\
& \wedge Before\ (c, d) \\
& \wedge (\forall e)(e \in T(R_1, ..., R_k, w) \Longrightarrow Before\ (e, c) \vee Equal\ (e, c) \\
& \qquad \vee Before\ (d, e) \vee Equal\ (d, e))
\end{aligned}$$

In this predicate, the last line means that there is no event in the time between c and d. The constant predicate will allow us to treat each constant time interval $[c, d)$ separately, thus reducing the inner query to a number of queries, each dealing with a constant time interval. In other words, we will be able to follow the same steps as in the snapshot Quel case. For each time interval $[c, d)$ given by the constant predicate a value of the aggregate, valid from c to d, will be computed and will potentially go into the result. This value is guaranteed to be unique by the definition of *Constant*.

EXAMPLE. For the *Faculty* relation, only for the following values of c and d is the *Constant*(*Faculty*, c, d, 0) predicate true (implying **for each instant**):

c	*d*
0	9-71
9-71	9-75
9-75	12-76
12-76	9-77
9-77	11-80
11-80	12-80
12-80	12-82
12-82	12-83
12-83	∞

Note that these consecutive intervals are exactly the ones indicated in Figure 1. For a moving window of `for each quarter`, we would use the window function $w(t) = 2$, resulting in

c	*d*
0	9-71
9-71	9-75
9-75	12-76
12-76	2-77
2-77	9-77
9-77	11-80
11-80	12-80
12-80	1-81
1-81	2-81
2-81	12-82
12-82	2-83
2-83	12-83
12-83	2-84
2-84	∞

||||||

3.4. Aggregates in the Target List

For a multi-relational query with one aggregate in the target list, we will take the approach used in the Quel semantics: tuples from the aggregate operation will be computed first via a partitioning function. Again, let F be any of the aggregate operators defined so far. Consider the TQuel query with one aggregate function in the target list,

```
range of $t_1$ is $R_1$
. . .
range of $t_k$ is $R_k$
retrieve ($t_{i_1}.D_{j_1}$, ..., $t_{i_r}.D_{j_r}$, $y$ = F($t_{l_1}.D_{m_1}$ by $t_{l_2}.D_{m_2}$, ..., $t_{l_n}.D_{m_n}$
                                                      for $\omega$
                                                      where $\psi_1$
                                                      when $\tau_1$
                                                      as of $\alpha_1$ through $\beta_1$ ))
    valid from $\upsilon$ to $\chi$
    where $\psi$
    when $\tau$
    as of $\alpha$ through $\beta$
```

in which

$$
\begin{array}{l}
1 \le i_1 \le k, \ldots, 1 \le i_r \le k \\
1 \le l_1 \le k, \ldots, 1 \le l_n \le k \\
1 \le j_1 \le deg(R_{i_1}), \ldots, 1 \le j_r \le deg(R_{i_r}) \\
1 \le m_1 \le deg(R_{l_1}), \ldots, 1 \le m_n \le deg(R_{l_n}).
\end{array}
$$

As with Quel, the where predicate should refer only to the tuple variable t_{l_1} or the tuple variables appearing in the by clause. The same restriction holds for the when clause appearing in the aggregate (no tuple variables are permitted in the as-of clause).

Here, the partitioning function will be based upon the four clauses that modify the aggregate (the by, where, when, and as-of clauses). Hence, using the same notation as in Section 1.3,

$$
\begin{array}{lll}
1 & P\ (a_2, \ldots, a_n, c, d) \triangleq \Big\{ b^{(p)} \mid (\exists t_{l_1}) \cdots (\exists t_{l_n}) & \\
2 & & (R_{l_1}(t_{l_1}) \land \cdots \land R_{l_n}(t_{l_n}) \\
3 & & \land\, b[1] = t_{l_1}[1] \land \cdots \land b[p] = t_{l_n}[deg(R_{l_n})] \\
4 & & \land\, t_{l_2}[m_2] = a_2 \land \cdots \land t_{l_n}[m_n] = a_n \\
5 & & \land\, \psi_1{}' \\
6 & & \land\, \Gamma_{\tau_1} \\
7 & & \land\, (\forall h)(1 \le h \le n)\ overlap\,([\Phi_{\alpha_1}, \Phi_{\beta_1}), [t_{i_h}[start], t_{i_h}[stop]]) \\
8 & & \land\, (\forall h)(1 \le h \le n)\ overlap\,([c, d), [t_{l_h}[from], t_{l_h}[to] + \omega'(c))) \\
9 & &) \Big\}
\end{array}
$$

where c and d are valid times, with $Before(c, d)$ and $p = \left[\sum_{i=1}^{n}(deg(R_{l_i})\right]+4$ (p includes the implicit attributes of t_{l_1} only). This definition assumes that the tuple variables t_{l_1}, ..., t_{l_n} are distinct. If they are not, then the duplicate tuple variables should be removed. In comparing this with the Quel partitioning function, notice that there are three additional lines here. Line 6 translates the when clause. Line 7 translates the as-of clause, specifying that the transaction times of all tuples of the inner query, including those in the inner where and when clauses, must overlap the rollback time specified in the as-of clause. This is similar to the as-of line in the outer query in TQuel. The window function ω' in line 8 corresponds to the keyword ω found in the retrieve statement. Line 8 indicates that all tuples participating in the aggregate must overlap the interval $[c, d)$ (from the definition of the *Constant* predicate, which will supply the intervals $[c, d)$, it is not difficult to see that the overlapping is total.) This way, aggregates will always be computed from the tuples that were valid during that interval. In determining the overlap, the window function ω' is used in a similar fashion to the definition of the time partition. If R_{l_k} in line 8 is an event relation, the predicate should be

$$overlap\,([c, d), [t_{l_k}[at], t_{l_k}[at]+\omega'(c)))$$

The output relation from a query with a single aggregate in the target list is

$$\begin{array}{ll}
1 & \Big\{ w^{((r+1)+4)} \mid (\exists t_1) \cdots (\exists t_k)(\exists c)(\exists d) \\
2 & \quad (R_1(t_1) \wedge \cdots \wedge R_k(t_k) \wedge \mathit{Constant}(R_{l_1}, \ldots, R_{l_n}, c, d, \omega') \\
3 & \quad \wedge (\forall l_i)(1 \le i \le n)(\mathit{overlap}([c, d), [t_{l_i}[\mathit{from}], t_{l_i}[\mathit{to}]))) \\
4 & \quad \wedge w[1] = t_{i_1}[j_1] \wedge \cdots \wedge w[r] = t_{i_r}[j_r] \\
5 & \quad \wedge w[r+1] = F(P(t_{l_1}[m_2], \ldots, t_{l_n}[m_n], c, d))[m_1] \\
6 & \quad \wedge w[r+2] = \mathit{last}(c, \Phi_\upsilon) \wedge w[r+3] = \mathit{first}(d, \Phi_\chi) \wedge \mathit{Before}(w[r+2], w[r+3]) \\
7 & \quad \wedge w[r+4] = \textit{current transaction time} \wedge w[r+5] = \infty \\
8 & \quad \wedge \psi' \\
9 & \quad \wedge \Gamma_\tau \\
10 & \quad \wedge (\forall l)(1 \le l \le k)(\mathit{overlap}([\Phi_\alpha, \Phi_\beta), [t_l[\mathit{start}], t_l[\mathit{stop}]))) \\
11 & \quad) \Big\}
\end{array}$$

A comparison with the tuple calculus expression given in Section 3.1 reveals that lines three and five are new and lines one and six are altered. The *Constant* predicate determines the interval [c, d) during which the tuples are constant. It involves the relations appearing in the aggregate; the relation whose attribute is being aggregated plus all the different relations in the by-list; other relations cannot affect the aggregate. Again, these relations are assumed to be distinct for notational convenience. The window function ω' appears explicitly as an argument to the *Constant* predicate and implicitly in P. Line three ensures that the tuple variables aggregated over and those specified in the by clause overlap with the interval during which the aggregate is constant. Line five computes the aggregate. Line six ensures that the valid time of the result relation is the intersection with the specified valid time and the interval [c, d). Two slight modifications are required for special cases. If the **valid at** υ variant is used, line 6 should be replaced with

$$w[r+2] = \Phi_\upsilon \wedge \mathit{overlap}([c, d), [w[r+2], w[r+2]+1))$$

Secondly, as with the Quel semantics, if t_{l_1} does not appear outside of the aggregate or in the by clause, it should also not appear in lines 1 and 2 (it *will* appear in the *Constant* predicate). Also, tuple variables mentioned in the aggregate that do not appear outside the aggregate should not appear in line 3.

EXAMPLE. Let us translate Example 6 into the tuple calculus.

$$
\begin{aligned}
P(a_2, c, d) \triangleq \Big\{ b^{(3+2)} \mid & (\exists f) \\
& (Faculty(f) \\
& \wedge b = f \\
& \wedge f[Rank] = a_2 \\
& \wedge overlap([c, d), [f[from], f[to]+0)) \\
&) \Big\}
\end{aligned}
$$

A window size of 0 is used because the default is **`for each instant`**. Some instances of the values of this function are

P(Assistant, 9-71, 9-75) = {(Jane, Assistant, 25000, 9-71, 12-76)}

P(Assistant, 9-75, 12-76) = {(Jane, Assistant, 25000, 9-71, 12-76),
(Tom, Assistant, 23000, 9-75, 12-80)}

The output relation is

$$
\begin{aligned}
\Big\{ w^{(2+2)} \mid & (\exists f)(\exists c)(\exists d) \\
& (Faculty(f) \wedge Constant(Faculty, c, d, 0) \\
& \wedge overlap([c, d), [f[from], f[to])) \\
& \wedge w[1] = f[Rank] \\
& \wedge w[2] = count(P(f[Rank], c, d)[Name] \\
& \wedge w[3] = last(c, f[from]) \wedge w[4] = first(d, f[to]) \wedge Before(w[3], w[4]) \\
& \wedge overlap([f[from], f[to]), [now, now+1)) \\
&) \Big\}
\end{aligned}
$$

The last two lines correspond to the default valid and when clauses. ||||||

For an aggregate with no by-list, only the where, when, and as-of clauses may be present, and the partitioning function P becomes again a subset of R_{l_1} :

$$
\begin{aligned}
P\,(c\,,d) = \Big\{ b^{(\rho)} \mid (R_1(b) \wedge \psi_1{}' \wedge \Gamma_{\tau_1} \\
&\wedge Before\,(b\,[start\,],\Phi_{\beta_1}) \wedge Before\,(\Phi_{\alpha_1}, b\,[stop\,])) \\
&\wedge overlap\,([c\,,d\,), [b\,[from\,], b\,[to\,])) \\
&) \Big\}
\end{aligned}
$$

The tuple calculus statement for the query remains the same as above, except that $P\ (c\,,d\,)$ is used in place of $P\ (t_{l_1}[m_2], \ldots, t_{l_k}[m_n], c\,, d\,)$ and only R_{l_1}, c, and d are needed as arguments to the *Constant* predicate.

The semantics is changed only slightly if either the underlying or result relations are event relations.

EXAMPLE. This is the tuple calculus version of Example 14 from Section 2.4.

$$
\begin{aligned}
P\,(c\,,d\,) \triangleq \Big\{ b^{(1+1)} \mid (\exists x)\ experiment\,(x) \\
&\wedge b = x \\
&\wedge overlap\,([c\,,d\,), [x\,[from], x\,[to]+\infty)) \\
&\Big\}
\end{aligned}
$$

$$
\begin{aligned}
\Big\{ w^{(2+1)} \mid (\exists x)(\exists c\,)(\exists d\,) \\
&(experiment\,(x\,) \wedge Constant\,(experiment\,, c\,, d\,, \infty) \\
&\wedge overlap\,([c\,, d\,), [x\,[at\,], x\,[at\,]+1)) \\
&\wedge w\,[1] = varts\,(P\,(c\,,d\,)) \wedge w\,[2] = 119 \cdot avgti\,(P\,(c\,,d\,))[Yield\,] \\
&\wedge w\,[3] = x\,[at\,] \wedge overlap\,([c\,,d\,), [w\,[3], w\,[3]+1)) \\
&) \Big\}
\end{aligned}
$$

The multiplier of *avgti* is discussed in Section 2.2. ||||||

3.5. Unique Aggregation

Unique aggregation is also possible in TQuel. There are four unique aggregates: `countU`, `sumU`, `avgU`, and `stdevU`. It is not necessary to define unique versions for `any`, `max`, `min`, `first`, `last`, `avgti` and `varts`, because the same results can be obtained with the non-unique aggregates.

As in Quel (c.f., Section 1.4), the semantics of unique aggregation utilizes an additional partitioning function U defined in terms of the original partitioning function P. When the inner query has a by-list, the modified partitioning function is defined in terms of the ordinary P as

$$U(a_2, ..., a_n, c, d) = \left\{ u^{(1)} \mid (\exists b)(b \in P(a_2, ..., a_n, c, d) \wedge u[1] = b[m_1]) \right\}$$

With no by-list, the modified partitioning function $U(c, d)$ is similarly defined from $P(c, d)$. In either case, only the explicit attributes remain in U; the implicit time attributes are not copied into U. The simple substitution of U for P in the final tuple calculus statement, together with the use of the non-unique versions of the aggregates, yields the tuple calculus semantics of unique aggregates.

EXAMPLE. The partitioning function for Example 13 is

$$\begin{aligned} P(c, d) = \Big\{ b^{(3+2)} \mid (\exists f) \\ &(Faculty(f) \\ &\wedge b = f \\ &\wedge Before(f[from], "1981"[from]) \\ &\wedge overlap([c, d), [f[from], f[to]+\infty) \\ &)\Big\} \end{aligned}$$

$$U(c, d) = \left\{ u^{(1)} \mid (\exists b)(b \in P(c, d) \wedge u[1] = b[Salary]) \right\} \quad ||||||$$

3.6. Multiple Aggregation

A TQuel query may call for several aggregates, some of them instantaneous and some others cumulative, potentially over different window sizes. Of course, each of the aggregates is computed from its own partitioning functions, using the appropriate window size. The *Constant* predicate is replaced by a similar predicate employing multiple time-partitions T_i, $1 \le i \le n$, each associated with one of the n aggregates:

$$
\begin{array}{l}
(\exists i, 1 \le i \le n)(\exists j, 1 \le j \le n)(\\
\qquad\qquad c \in T_i(R_1, \ldots, R_k, w_i) \\
\qquad\qquad \wedge\, d \in T_j(R_1, \ldots, R_k, w_j) \\
\qquad\qquad \wedge\, c \ne d \\
\qquad\qquad \wedge\, \mathit{Before}\,(c, d) \\
\qquad\qquad \wedge\, (\forall e)(\forall l, 1 \le l \le n) \\
\qquad\qquad\qquad\qquad (e \in T(R_1, \ldots, R_k, w_l) \Longrightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathit{Before}\,(e, c) \vee \mathit{Equal}\,(e, c) \vee \mathit{Before}\,(d, e) \vee \mathit{Equal}\,(d, e)) \\
)
\end{array}
$$

Each T_i can either range over all of the relations appearing in any aggregate, or can range over only those relations appearing in the specific aggregate associated with T_i. When there is only one aggregate, this predicate is identical to the *Constant* predicate.

Valid times for each output tuple are computed by following the same approach as before: each output tuple is valid during an interval when tuples from all the non-aggregate attributes are in the $[\Phi_\upsilon, \Phi_\chi)$ interval, and this interval overlaps the valid times of the calculated aggregates.

3.7. Aggregates in the Outer Where Clause

TQuel aggregates, or arithmetic expressions containing TQuel aggregates, may be part of the main where or when clause. Through the partitioning functions, the values of the aggregated attribute are first computed, then used in place of the aggregate in the predicate of the query. Since the variables in by-lists are "global", its by clause is linked to the rest of the query, as in Quel.

3.8. Nested Aggregation

In nested aggregation, the local where clause of an aggregate F_1 invokes another aggregate F_2. If F_2 has a by-list, links are established between the tuple variables in the by-list of F_2 and the tuple variables in the F_1 query. The *Constant* predicate in the retrieve statement is replaced with the predicate over multiple time partitions given in Section 3.6.

EXAMPLE. Example 11 contains a nested aggregate. Let us show the partitioning functions P_1 and P_2 for the outer and the inner aggregates respectively:

$$P_1(c, d) = \Big\{ b^{(3+2)} \mid (\exists f)\,(Faculty(f)$$

$$\wedge\, b = f$$

$$\wedge\, f\,[Salary] \neq min(P_2(c, d))[Salary]$$

$$\wedge\, overlap\,([c, d), [f\,[from], f\,[to]{+}0))$$

$$)\Big\}$$

$$P_2(c, d) = \Big\{ b^{(3+2)} \mid (\exists f)\,(Faculty(f)$$

$$\wedge\, b = f$$

$$\wedge\, overlap\,([c, d), [f\,[from], f\,[to]{+}0))$$

$$)\Big\}$$

The tuple calculus statement for the retrieve statement will contain $P_1(c, d)$; $P_2(c, d)$ only occurs within P_1. In this case, both aggregates were `for each instant`. Different window functions are accommodated by using the appropriate window function in each partitioning function, and by referencing all window functions in the predicate replacing *Constant*. ||||||

3.9. Aggregates in the Other Outer Clauses

Two aggregates may be used in the when and valid clauses: `earliest` and `latest`. Just like in the case of aggregates in the where clause, an aggregate that is used in the when clause can be modified with inner by, for, where, when, and as-of clauses. With these restrictions, the semantics of the aggregated temporal constructors is the same as that of the other aggregates. For the linking of tuple variables, the same comments in sections 3.7 and 3.8 concerning the outer and inner where clause apply. Being based on *first* and *last* (c.f., Section 3.1), there is no need to define unique versions of the aggregated temporal constructors.

EXAMPLE. Example 12 illustrates this point.

$$P(a_2, c, d) \triangleq \left\{ b^{(3+2)} \mid (\exists f)\,(Faculty(f) \right.$$

$$\land\, b = f$$

$$\land\, f[Rank] = a_2$$

$$\land\, overlap([c, d), [f[from], f[to]+\infty))$$

$$\left. \right\}$$

P(Assistant, 9-71, 9-75) = {(Jane, Assistant, 25000, 9-71, 12-76)}

The relation resulting from the query is

$$\left\{ w^{(2+2)} \mid (\exists f)(\exists c)(\exists d) \right.$$

$$(Faculty(f) \land Constant(Faculty, c, d, \infty)$$

$$\land\, overlap([c, d), [f[from], f[to]))$$

$$\land\, w[1] = f[Name] \land w[2] = f[Rank]$$

$$\land\, w[3] = last(c, f[from]) \land w[4] = first(d, f[to]) \land Before(w[3], w[4])$$

$$\land\, Before(earliest(P(f[Rank], c, d))[from], f[from])$$

$$\land\, Before(f[from], earliest(P(f[Rank], c, d))[to])$$

$$\left.) \right\}$$

The fifth line originates from the default valid clause, which in this case is **`valid from begin of`** `f` **`to end of`** `f`. That the cumulative version of the aggregate was specified in the TQuel query is reflected in a window size of ∞. ||||||

4. Related Work

As was mentioned in the introduction, most conventional query languages include support for aggregates. There has also been some formal work on aggregates. Klug introduced an approach to handle aggregates within the formalism of both relational algebra and tuple relational calculus [Klug 1982]. His method makes it possible to define both standard and unique aggregates in a rigorous way. Ceri and Gottlob present a translation from a subset of SQL that includes aggregates into relational algebra, thereby defining an operational semantics for SQL aggregates [Ceri & Gottlob 1985]. Also, significant progress has been made in the area of *statistical databases* [LBL 1981, LBL 1983]. Such databases, used primarily for summary statistics gathering and statistical analysis, contain set-valued attributes. Klug's relational algebra and calculus have been extended to manipulate set-valued attributes and to utilize aggregate functions [Ozsoyoglu, et al. 1986], thereby forming a theoretical framework for statistical database query languages.

Aggregates may also be found in several of the dozen query languages supporting time that have appeared over the last decade. In some of these languages, aggregates play only a small role. Ben-Zvi included several aggregate operators and functions in his TRM language, although not in a very clear or comprehensive manner [Ben-Zvi 1982]; Ariav also mentioned aggregates in the context of his TOSQL language [Ariav 1985]. Although Gadia's HTQuel language does not explicitly include aggregates, his "temporal navigation" operators (e.g., First) can be simulated using aggregates, since they effectively extract an interval from a collection of intervals [Gadia & Vaishnav 1985].

Finally, four other languages supporting time include a comprehensive set of aggregates and associated constructs. Legol 2.0 was one of the first time-oriented query languages to appear [Jones et al. 1979]. This language is based on the relational algebra. HQuel, an extension of Quel, is based on a model incorporating set-valued, time-stamped attributes [Tansel & Arkun 1986]. TSQL is an extension of SQL [IBM 1981] incorporating valid time [Navathe & Ahmed 1986]. The operations over the time sequence collections of the temporal data model (*TDM*), presented in an SQL-like syntax, include AGGREGATE and ACCUMULATE statements [Segev & Shoshani 1987].

In the remainder of this section, these three query languages will be compared with Quel and TQuel against a set of criteria. These eighteen criteria were chosen because they are well-defined, are independent of any specific query language, and are demonstrably beneficial. Table 1 summarizes the comparison.

Table 1: Comparison of Query Languages Supporting Time

Criterion	TQuel	Quel	Legol	HQuel	TSQL	TDM
Formal Semantics Provided	√	√	□	□	□	□
Aggregates in Outer Selection	√	√	√	?	√	P
Selection within Aggregates	√	√	√	?	√	□
Aggregates on Partitions	√	√	√	?	√	√
Nested Aggregation	√	√	√	?	√	□
Multiple-relation Aggregates	√	√	√	√	√	√
Operational Semantics Provided	√	√	√	√	□	□
Implementation Exists	□	√	?	□	□	□
Unique and Non-unique Aggregation	√	√	□	?	√	√
Temporal Partitioning	P	–	□	□	√	√
Temporal Selection Within Agg. Over Valid Time	√	–	√	?	√	√
Temporal Selection Within Agg. Over Trans. Time	√	–	□	□	□	□
Aggregates In Outer Temporal Selection	√	–	√	?	√	□
Instantaneous Aggregates	√	–	√	□	P	P
Cumulative Aggregates	√	–	√	√	√	√
Moving-window Aggregates	√	–	□	□	√	?
Temporally Weighted Aggregates	√	–	□	√	□	□
Aggregates over Chronological Order	√	–	√	√	√	√

√ satisfies criterion
P partial compliance
□ criterion not satisfied
? not specified in papers
– not applicable

These criteria arise from three sources, the first being aspects that apply to most conventional aggregates, and hence should be satisfied by proposed temporal aggregates.

• The aggregates should have a *formal semantics.* Without a formal definition, the meaning of each construct, and the interaction between constructs, is unclear. Only Quel and TQuel aggregates have been formally specified, both in this paper.

• *Aggregates in the outer selection* should be supported. Quel, TQuel, TSQL and perhaps HQuel (the feature isn't mentioned in the paper) permit aggregates in the selection construct, in this case, the where clause. Legol permits aggregates in any expression, including comparisons. TDM permits a very limited collection of aggregates in the where clause.

• *Selection within aggregates* should be supported. Quel allows a where clause within an aggregate to specify a subset of tuples over which the aggregate is computed. TQuel also permits such a where clause, and HQuel may. Legol allows aggregates to be computed over any relational expression. TSQL introduces a new construct, HAVING, to specify nested selection. TDM doesn't allow a where clause in the AGGREGATE or ACCUMULATE statements.

• *Aggregation on partitions* should be possible. Quel, TQuel, and HQuel use the by clause, and TSQL and TDM use the GROUP BY clause, to specify partitioned aggregation. Legol does not include such a construct.

• *Nested aggregation* should be supported. Quel, TQuel, TSQL, Legol, and perhaps HQuel support aggregates whose arguments are themselves aggregates; TDM does not.

• *Multiple-relation aggregates* should be supported. Quel, TQuel, and HQuel permit several tuple variables to appear in an aggregate. Legol, TSQL and TDM perform aggregation over arbitrary expressions, thereby accommodating multiple relations.

• Aggregates should have a well-defined *operational semantics*. By this we mean that a formal temporal algebra including aggregates should be defined, and a mapping of aggregates in the language to algebraic expressions should be provided. Klug showed how aggregates can be handled within the relational algebra and calculus [Klug 1982]; his approach can be applied to Quel to satisfy this criteria. A temporal relational algebra has been defined that supports the TQuel aggregates, including aggregates in the target list, inner where and when clauses, and outer where, when, and valid clauses [McKenzie & Snodgrass 1987A, McKenzie & Snodgrass 1987B]. A different algebra supports HQuel's aggregates [Tansel 1986]. Legol is itself an algebra. While an algebra is defined that supports TSQL, this algebra does not include aggregates [Navathe & Ahmed 1986]. TDM does not have an equivalent algebra.

• An *implementation* of the aggregates should exist. Quel aggregates have been implemented in the Ingres DBMS. An early version of Legol has been implemented, but it is not stated whether aggregates were implemented in this prototype. No other proposal has been implemented.

• *Unique and non-unique aggregation* should be supported. The latter is useful to avoid incurring the overhead of sorting the relation before aggregation to remove duplicates. Quel, TQuel, TSQL, TDM and perhaps HQuel support both unique and non-unique aggregation. It appears that Legol supports only unique aggregation.

The second source of evaluation criteria are aspects of conventional aggregates that can be extended in an obvious fashion to apply to time.

• *Temporal partitioning* should be supported. This feature, analogous to aggregates partitioned on the explicit attributes, was also first introduced in TSQL via the GROUP BY construct [Navathe & Ahmed 1986]. It is similar to the moving window (see below), except that the window is fixed. TDM has an analogous GROUP T BY construct. This feature can be simulated in TQuel by using auxiliary relations, as discussed in section 2.4. No other query language supports temporal partitioning.

• *Temporal selection within aggregates over valid time* should be supported. This feature, analogous to conventional selection within aggregates, is supported in TQuel and TSQL via a when clause that specifies a subset of tuples, based on when the tuples were valid, over which the aggregate is computed. Legol allows aggregates to be computed over any relational expression. HQuel may, and TDM does, support temporal selection through temporal operators in the where clause.

• *Temporal selection within aggregates over transaction time* should be supported. While only TQuel supports this feature, it appears that an as-of clause could be added to the other languages fairly easily [McKenzie & Snodgrass 1987C].

• *Aggregates in the outer temporal selection* should be supported. Again, this is analogous to supporting aggregates in the outer conventional selection. TQuel, TSQL, and Legol support this feature. HQuel does not include any aggregate operators that operate directly on time stamps. In TDM, AGGREGATE is a separate statement from SELECT.

The final source of evaluation criteria are previous papers on aggregates that introduce desirable features. The relative importance of these and other potential features will emerge only with further work in this area.

• *Instantaneous aggregates* should be supported. These aggregates yield a distribution on the time axis, where the value of the aggregate at instant *t* is computed from tuples valid at time *t* [Jones et al. 1979]. Both Legol and TQuel support such aggregates. They can be approximated in TSQL and TDM by using a very small moving window. Instantaneous aggregates cannot be specified in HQuel.

• *Cumulative aggregates* should be supported. These aggregates compute a value at each time *t* from tuples valid at or before time *t* [Jones et al. 1979]. In this comparison we differentiate between strict cumulative aggregates, as defined by Jones, et al., and moving window aggregates, as defined by Navathe and Ahmed. TQuel, Legol, TSQL, and TDM support cumulative aggregates. In HQuel, all aggregates are cumulative.

• *Moving-window aggregates* should be supported. These aggregates compute a value at each time *t* from tuples valid sometime during the specified window interval ending at *t* [Navathe & Ahmed 1986]. This feature was originally termed "moving time-window". TQuel and TSQL fully support moving-window aggregates; TDM may support moving-window aggregates through the GROUP T BY construct; and the other query languages do not support this language feature.

• *Temporally weighted aggregates* should be available. Tansel introduced the concept of an average weighted by the duration of the values [Tansel & Arkun 1986]; the concept was also briefly mentioned elsewhere [Snodgrass 1982]. TQuel's `avgti` aggregate serves a similar purpose. The other languages doe not provide such aggregates.

• *Aggregates over the chronological order of tuples* should be available. The first and last aggregates of Legol have been included in the other languages.

In summary, TQuel's aggregates meet all but one criteria (the exception being an implementation); the other query languages are all lacking in several criteria.

5. Conclusion

This paper makes three contributions. First, a formal semantics for the conventional query language Quel was presented. This completes the formal definition of Quel (the core of the retrieve statement and the modification statements were previously formalized [Snodgrass 1987, Ullman 1982]).

Secondly, the aggregates in Quel have been extended in a minimal fashion for inclusion in TQuel. TQuel added the when and as-of clauses, which are the temporal analogues for valid and transaction time, respectively, to the where clause. These clauses are permitted within the aggregate. The other syntactic extension is the for clause, used to distinguish between instantaneous, cumulative, and moving window aggregates. Additional temporal aggregate operators were also introduced.

Finally, the Quel tuple calculus semantics was extended to accommodate time-varying relations. Our approach used the *Constant* predicate and a partitioning function to determine those intervals over which a relation remains static, enabling the aggregate value to be computed in a conventional manner.

The result is a complete formal semantics for TQuel and its snapshot subset Quel. A complete formal semantics for no other relational query language, temporal or otherwise, has been defined.

Acknowledgements

We are grateful to Peter Bloomfield for his remarks on the requirements of experimental data in statistical time series that lead to the creation of the `varts` operator, and to Ilsoo Ahn, David Beard, and Juan Valiente for helpful comments on this paper. This work was supported in part by NSF grant DCR-8402339, by ONR contract N00014-86-K-0680, and by a Junior Faculty Development Award from the UNC-CH Foundation. The first author was supported in part by an IBM Faculty Development Award. The third author was sponsored in part by the U.S. Air Force.

References

[Ariav 1985] Ariav, G. *A Temporally Oriented Data Model.* Technical Report. New York University. Mar. 1985.

[Ben-Zvi 1982] Ben-Zvi, J. *The Time Relational Model.* PhD. Diss. Computer Science Department, UCLA, 1982.

[Ceri & Gottlob 1985] Ceri, S. and G. Gottlob. *Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. IEEE Transactions on Software Engineering,* SE-11, No. 4, Apr. 1985, pp. 324-345.

[Codd 1972] Codd, E. F. *Relational Completeness of Data Base Sublanguages,* in Data Base Systems. Vol. 6 of Courant Computer Symposia Series. Englewood Cliffs, N.J.: Prentice Hall, 1972. pp. 65-98 .

[Date 1983] Date, C. J. *An Introduction to Database Systems.* Vol. II of Addison-Wesley Systems Programming Series. Reading, MA: Addison-Wesley Pub. Co., Inc., 1983.

[Epstein 1979] Epstein, R. *Techniques for Processing of Aggregates in Relational Database Systems.* UCB/ERL M7918. Computer Science Department, University of California at Berkeley. Feb. 1979.

[Gadia & Vaishnav 1985] Gadia, S.K. and J.H. Vaishnav. *A Query Language for a Homogeneous Temporal Database,* in *Proceedings of the ACM Symposium on Principles of Database Systems,* Apr. 1985.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES--A Relational Data Base Management System. Proceedings of the AFIPS 1975 National Computer Conference,* 44, May 1975, pp. 409-416.

[IBM 1981] IBM *SQL/Data-System, Concepts and Facilities.* Technical Report GH24-5013-0. IBM. Jan. 1981.

[Jones et al. 1979] Jones, S., P. Mason and R. Stamper. *LEGOL 2.0: A Relational Specification Language for Complex Rules. Information Systems,* 4, No. 4, Nov. 1979, pp. 293-305.

[Klug 1982] Klug, A. *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. Journal of the Association of Computing Machinery,* 29, No. 3, July 1982, pp. 699-717.

[McKenzie & Snodgrass 1987A] McKenzie, E. and R. Snodgrass. *Scheme Evolution and the Relational Algebra.* Technical Report TR87-003. Computer Science Department, University of North Carolina at Chapel Hill. May 1987.

[McKenzie & Snodgrass 1987B] McKenzie, E. and R. Snodgrass. *Supporting Valid Time: An Historical Algebra.* Technical Report TR87-008. Computer Science Department, University of North Carolina at Chapel Hill. Aug. 1987.

[McKenzie & Snodgrass 1987C] McKenzie, E. and R. Snodgrass. *Extending the Relational Algebra to Support Transaction Time,* in *Proceedings of ACM SIGMOD International Conference on Management of Data,* Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: May 1987, pp. 467-478.

[Navathe & Ahmed 1986] Navathe, S.B. and R. Ahmed. *A Temporal Relational Model and a Query Language.* UF-CIS Technical Report TR-85-16. Computer and Information Sciences Department, University of Florida. Apr. 1986.

[Ozsoyoglu, et al. 1986] Ozsoyoglu, G., Z.M. Ozsoyoglu and V. Matos. *Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions.* Technical Report. Department of Computer Engineering and Science, Case Western Reserve University. 1986.

[Segev & Shoshani 1987] Segev, A. and A. Shoshani. *Logical Modeling of Temporal Data*, in *Proceedings of the SIGMod 1987 Annual Conference*, Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: ACM Press, May 1987, pp. 454-467.

[Snodgrass 1982] Snodgrass, R. *Monitoring Distributed Systems: A Relational Approach.* PhD. Diss. Computer Science Department, Carnegie-Mellon University, Dec. 1982.

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases. IEEE Computer,* 19, No. 9, Sep. 1986, pp. 35-42.

[Snodgrass 1987] Snodgrass, R. *The Temporal Query Language TQuel. ACM Transactions on Database Systems,* 12, No. 2, June 1987, pp. 247-298.

[Stonebraker et al. 1976] Stonebraker, M., E. Wong, P. Kreps and G. Held. *The Design and Implementation of INGRES. ACM Transactions on Database Systems,* 1, No. 3, Sep. 1976, pp. 189-222.

[Tansel 1986] Tansel, A.U. *Adding Time Dimension to Relational Model and Extending Relational Algebra. Information Systems,* 11, No. 4 (1986), pp. 343-355.

[Tansel & Arkun 1986] Tansel, A.U. and M.E. Arkun. *HQUEL, A Query Language for Historical Relational Databases.* Technical Report. Bernard M. Baruch College, CUNY. Jan. 1986.

[Ullman 1982] Ullman, J.D. *Principles of Database Systems, Second Edition.* Potomac, Maryland: Computer Science Press, 1982.

[LBL 1981] *Proceedings of the First International Workshop on Statistical Database Management.* Ed. H.K. Wong. 1981.

[LBL 1983] *Proceedings of the Second International Workshop on Statistical Database Management.* Ed. J. McCarthy. 1983.

Appendix: Syntax Summary

In order to accommodate aggregates, the TQuel syntax [Snodgrass 1987] is slightly augmented. TQuel is a superset of Quel, that is, all legal Quel statements with aggregates are also legal TQuel statements with aggregates. The following are the additions made to the above mentioned TQuel syntax.

```
< expression>         ::= In addition to the TQuel syntax, include:
                        | < aggregate term>
< aggregate term>     ::= < aggregate op>  ( < expression> < aggregate tail>  )
                      ::= varts  ( < e-expression> < aggregate tail>  )
< aggregate tail>     ::= < by clause> < for clause> < retrieve tail>
< by clause>          ::= ε | by < attribute list>
< attribute list>     ::= < expression> | < attribute list> , < expression>
```

< aggregate op>	::= count | countU | sum | sumU | avg | avgU | stdev | stdevU
	| any | min | max | first | last | avgti **per** < time unit>
< for clause>	::= ε | **for each** < time unit> | **for each instant** | **for ever**
< time unit>	::= **millisecond** | **second** | **minute** | **hour**
	| **day** | **week** | **month** | **quarter** | **year** | **decade** | · · ·
< interval element>	::= *In addition to the TQuel syntax, include:*
	| < aggt> (< i-expression> < aggregate tail>)
< aggt>	::= earliest | latest

where < i-expression> evaluates to an interval (i.e., a pair of timestamps) and < e-expression> evaluates to an event (i.e., a single timestamp).